2021.11.20. 교육

나정휘
https://justiceHui.github.io/

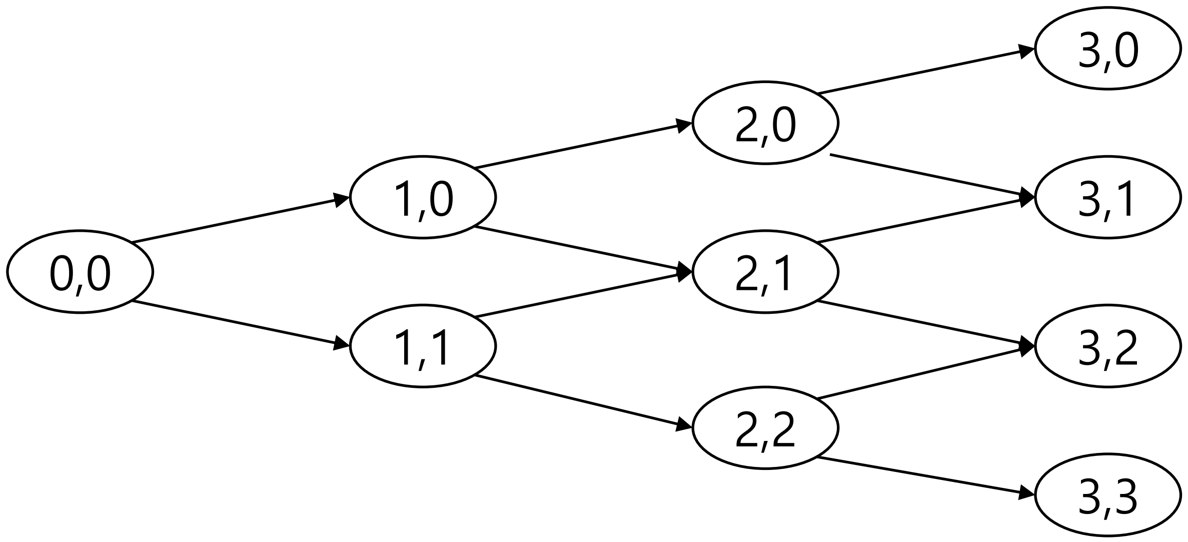
목차

- 트리에서 DP
- DAG에서 DP
- 구간에 대한 DP
- 경우의 수 / 확률 DP
- 비트필드를 이용한 DP
- Convex Hull Trick

트리 DP / DAG DP

- DP에서 부분 문제 간의 참조 관계는 DAG로 표현 가능
- 점화식을 DAG로 나타낸 뒤 위상 정렬하면서 점화식을 계산할 수 있음
  - ex) D(n, k) = D(n-1, k) + D(n-1, k-1)

- 트리는 재귀적인 구조 / 루트를 고정하면 DAG이므로 비슷하게 DP를 할 수 있음



BOJ 1949 우수 마을

- 각 정점은 우수 마을로 선정될 수도 있고, 선정되지 않을 수 있다.
  - 만약 우수 마을로 선정된다면, 이웃한 모든 정점은 우수 마을이 될 수 없음
  - 우수 마을로 선정되지 않는다면, 이웃한 정점 중 최소 한 정점은 우수 마을이 되어야 함

- 사실 3번 조건은 필요 없는 조건
  - 어떤 마을 v가 우수 마을이 아니면서 우수 마을과 인접하지도 않는 것이 최적해라면
  - v를 선택해서 정답을 더 늘릴 수 있으므로 모순

- D(v, 0) : v를 루트로 하는 서브 트리에서, v가 우수 마을로 선정되지 않았을 때의 최댓값
  - v의 자식 정점 c에 대해, D(v, 0) = sum{ max{ D(c, 0), D(c, 1) } }
- D(v, 1) : v를 루트로 하는 서브 트리에서, v가 우수 마을로 선정되었을 때의 최댓값
  - v의 자식 정점 c에 대해, D(v, 1) = A[v] + sum{ D(c, 0) }
- 시간 복잡도는 O(N)

BOJ 1949 우수 마을

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, A[10101], D[10101][2];
vector<int> G[10101];

void DFS(int v, int b=-1){
    for(auto i : G[v]){
        if(i == b) continue;
        DFS(i, v);
        D[v][0] += max(D[i][0], D[i][1]);
        D[v][1] += D[i][0];
    }
    D[v][1] += A[v];
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<N; i++){
        int s, e; cin >> s >> e;
        G[s].push_back(e), G[e].push_back(s);
    }
    DFS(1);
    cout << max(D[1][0], D[1][1]);
}
```

BOJ 20188 등산 마니아

- u에서 v로 갈 때, 산 정상을 거치는 가장 짧은 길
    - u -> LCA(u, v) -> 1 -> LCA(u, v) -> v
    - u에서 v로 가는 최단 경로와, 1과 LCA 사이의 경로로 분리해서 생각하는 것이 편함
    - u -> LCA(u, v) -> v / 1 <-> LCA(u, v)

- u에서 v로 가는 최단 경로
    - 각 간선이 몇 번 사용되는지 계산하자.
    - 부모 정점 p와 자식 정점 c를 연결하는 간선 (p, c)이 사용되는 횟수는 S[c] * (N – S[c])
        - S[v]는 v를 루트로 하는 서브 트리의 정점 개수
- 1과 LCA 사이의 경로
    - 각 정점이 LCA가 되는 경우의 수를 구하면, (경우의 수) * (정점의 깊이)를 구할 수 있다.
    - 정점 p의 자식을 c1, c2, … , ck라고 하면, p가 LCA가 되는 경우의 수는 C(S[p], 2) – sum{ C(S[ci], 2) }
        - (p를 루트로 하는 서브 트리에서 임의의 두 정점을 선택하는 경우) – (ci 밑에 있는 두 정점을 선택하는 경우)

BOJ 20188 등산 마니아

```cpp
#include <bits/stdc++.h>
using namespace std;
long long C2(int x){ return 1LL * x * (x - 1) / 2; }

// N, Size, Depth, Parent
int N, S[303030], D[303030], P[303030];
vector<int> G[303030];

void DFS(int v, int b=-1){
    S[v] = 1; P[v] = b;
    for(auto i : G[v]){
        if(i == b) continue;
        D[i] = D[v] + 1;
        DFS(i, v);
        S[v] += S[i];
    }
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<N; i++){
        int s, e; cin >> s >> e;
        G[s].push_back(e); G[e].push_back(s);
    }
    DFS(1);

    long long R = 0;
    for(int i=1; i<=N; i++) R += 1LL * S[i] * (N - S[i]);
    for(int i=1; i<=N; i++){
        long long now = C2(S[i]);
        for(auto j : G[i]) if(j != P[i]) now -= C2(S[j]);
        R += now * D[i];
    }
    cout << R;
}
```

BOJ 18780 Timeline

- S_i 조건이 없다고 생각하자.
- (a, b, x)라는 정보는 b가 a보다 최소 x일 이상 늦게 시작한다는 것을 의미한다.
  - a에서 b로 가는 가중치 x 간선을 만들자.
  - C개의 tuple이 나타내는 조건을 모두 만족하는 각 세션의 최소 날짜는 각 정점까지의 “최장 거리”와 동일하다.

- 그래프 모델링 아이디어를 잘 떠올렸다면, S_i 조건도 쉽게 처리할 수 있다.
  - 0일차에 열리는 0번 세션을 만들고
  - 0번 정점에서 i번 정점으로 가는 가중치 S_i 간선을 만들면 된다.

- 최장 거리를 구하는 것은 위상 정렬로 쉽게 할 수 있다.
- 시간 복잡도는 O(N+C)

BOJ 18780 Timeline

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
using PLL = pair<ll, ll>;

int N, M, K, A[101010], C[101010];
vector<PLL> G[101010];
ll D[101010];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> K;
    for(int i=1; i<=N; i++){
        cin >> A[i]; C[i]++;
        G[0].emplace_back(i, A[i]);
    }
    for(int i=1; i<=K; i++){
        int a, b, c; cin >> a >> b >> c;
        G[a].emplace_back(b, c); C[b]++;
    }

    queue<int> Q; Q.push(0);
    while(Q.size()){
        int v = Q.front(); Q.pop();
        for(auto [i,w] : G[v]){
            D[i] = max(D[i], D[v] + w);
            if(!--C[i]) Q.push(i);
        }
    }
    for(int i=1; i<=N; i++) cout << D[i] << "\n";
}
```

BOJ 16297 Eating Everything…

- D(v, c) = v번째 정점까지 c개의 피자를 먹었을 때 만족도의 최댓값
  - v, c는 최대 50만이므로 TLE/MLE
  - 0번 정점에서 출발하는 방식으로는 해결하기 어려워 보임.

- 위상 정렬의 역순으로 처리하면 편하게 할 수 있음
  - D[v] = v에서 출발했을 때 얻을 수 있는 만족도의 최댓값
    - v에서 피자를 안 먹으면 D[i]
    - v에서 피자를 먹으면 A[v] + D[i] / 2
  - v에서 i로 가는 간선이 있다고 하면, D[v] = max{ D[i], D[i] / 2 + A[v] }

- DFS가 끝나는 순서의 역순이 위상 정렬이므로, Tree DP처럼 DFS를 이용해 쉽게 구현할 수 있다.
- 시간 복잡도는 O(N+M)

BOJ 16297 Eating Everything…

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, M, C[505050];
vector<int> G[505050];
double A[505050], D[505050];

void DFS(int v){
    C[v] = 1; D[v] = A[v];
    for(auto i : G[v]){
        if(!C[i]) DFS(i);
        D[v] = max({ D[v], D[i], A[v] + D[i] / 2 });
    }
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=0; i<N; i++) cin >> A[i];
    for(int i=0; i<M; i++){
        int s, e; cin >> s >> e;
        G[s].push_back(e);
    }
    DFS(0);
    cout << fixed << setprecision(10) << D[0];
}
```

구간에 대한 DP

- “D(s, e) = s부터 e까지의 정답” 같은 느낌의 DP 문제가 꽤 많음
    - 문제마다 점화식을 채우는 방법도 다양함

- 인접한 두 구간을 합치는 방식으로 계산
    - 어떤 구간의 정답을 두 구간의 합으로 나타낼 수 있는 경우 (ex. 행렬 곱셈 순서)
    - D(i, j) = min{ D(i, k) + D(k+1, j) }
    - 시간 복잡도는 O(N^3), i < k < j이기 때문에 실제로는 N^3 / 6 정도 됨
    - 가끔 이상한 아이디어를 들고 와서 최적화를 하는 문제도 있음 (ex. BOJ 12008, BOJ 13974)
    - 너무 웰노운이라서 오늘은 안 함

- 한 칸 씩 구간을 확장하는 방식
    - 구간 [s, e]에서 [s-1, e]와 [s, e+1]로 이동하는 경우
    - 문제를 풀어보면서 살펴보자.

BOJ 2315 가로등 끄기

- 가로등을 끄는 시간은 0이므로 이동하면서 만나는 가로등은 끄는 것이 이득
  - [s, e] 구간의 가로등을 모두 껐다면 s 또는 e에 있는 상태
  - s-1로 이동해서 가로등을 끄거나, e+1로 이동해서 가로등을 끌 수 있음

- D(s, e, 0) : [s, e] 구간의 가로등을 모두 껐고 현재 왼쪽(s)에 있을 때, 낭비되는 전력의 최솟값
- D(s, e, 1) : [s, e] 구간의 가로등을 모두 껐고 현재 오른쪽(e)에 있을 때, 낭비되는 전력의 최솟값

- s-1로 이동하는 경우
  - D(s-1, e, 0) + (P[now] – P[s-1]) * (S[N] – S[e] + S[s-1])
    - P[i] : i번째 가로등의 위치, S[i] : 1..i번째 가로등의 전력 소모량(누적합)
- e+1로 이동하는 경우
  - D(s, e+1, 1) + (P[e+1] – P[now]) * (S[N] – S[e] + S[s-1])
- 시간 복잡도는 O(N^2)

BOJ 2315 가로등 끄기

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, M, P[1010], S[1010];
int D[1010][1010][2];

int f(int s, int e, int flag){
    if(s == 1 && e == N) return 0;
    int &res = D[s][e][flag];
    if(res != -1) return res;

    res = 0x3f3f3f3f;
    int now = flag ? e : s, on = S[N] - S[e] + S[s-1];
    if(s > 1) res = min(res, f(s-1, e, 0) + (P[now] - P[s-1]) * on);
    if(e < N) res = min(res, f(s, e+1, 1) + (P[e+1] - P[now]) * on);
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1; i<=N; i++){
        cin >> P[i] >> S[i];
        S[i] += S[i-1];
    }
    memset(D, -1, sizeof D);
    cout << f(M, M, 0);
}
```

BOJ 9520 NP-Hard

- 방문하는 정점의 번호는 감소하다가 특정 지점 이후부터는 증가하는 형태
  - ex) 5 3 2 1 4 6
  - 1만 있는 수열에서 시작해서, 2 3 4 …를 양옆에 붙이는 방식으로 DP

- D(s, e) = 가장 왼쪽에 s, 오른쪽에 e가 있을 때 남은 도시를 방문하는 최소 시간
  - 바로 다음에 붙일 도시의 번호 t = max(s, e) + 1
  - D(s, e) = min{ D(t, e) + C(s, t), D(s, t) + C(e, t) }
    - C(i, j) : i에서 j로 이동하는 비용

- 시간 복잡도는 O(N^2)

BOJ 9520 NP-Hard

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, C[1515][1515], D[1515][1515];

int f(int s, int e){
    if(max(s, e) == N) return 0;
    int &res = D[s][e];
    if(res != -1) return res;
    int x = max(s, e) + 1;
    return res = min(f(x, e) + C[s][x], f(s, x) + C[e][x]);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) for(int j=1; j<=N; j++) cin >> C[i][j];
    memset(D, -1, sizeof D);
    cout << f(1, 1);
}
```

확률 / 경우의 수 DP

- 정해진 유형은 없고, 그냥 머리를 열심히 써야 함
- 확률과 통계 공부를 열심히 하자!

- 알면 도움되는 것들
  - 피보나치 수
  - 카탈란 수
  - 교란 순열
  - 제1종/제2종 스털링 수
  - (DP는 아니지만) 번사이드 보조 정리

BOJ 1344 축구

- D(N, A, B) = N번째 간격에서 점수가 A:B일 확률
  - D(0, 0, 0) = 1, D(0, *, *) = 0

  - A만 득점하는 경우 : D(N-1, A-1, B) * Pa * (1 – Pa)
  - B만 득점하는 경우 : D(N-1, A, B-1) * Pb * (1 – Pb)
  - 모두 득점하는 경우 : D(N-1, A-1, B-1) * Pa * Pb
  - 모두 득점하지 못하는 경우 : D(N-1, A, B) * (1 – Pa) * (1 – Pb)

BOJ 1344 축구

```cpp
#include <bits/stdc++.h>
using namespace std;

bool IsPrime(int n){
    if(n < 2) return false;
    for(int i=2; i*i<=n; i++) if(n % i == 0) return false;
    return true;
}

double A, B, D[19][19][19], R;

int main(){
    cin >> A >> B;
    A /= 100; B /= 100;
    D[0][0][0] = 1;
    for(int i=1; i<=18; i++){
        for(int a=0; a<=i; a++){
            for(int b=0; b<=i; b++){
                if(a > 0) D[i][a][b] += D[i-1][a-1][b] * A * (1 - B);
                if(b > 0) D[i][a][b] += D[i-1][a][b-1] * (1 - A) * B;
                if(a > 0 && b > 0) D[i][a][b] += D[i-1][a-1][b-1] * A * B;
                D[i][a][b] += D[i-1][a][b] * (1 - A) * (1 - B);
            }
        }
    }
    for(int a=0; a<=18; a++){
        for(int b=0; b<=18; b++){
            if(IsPrime(a) || IsPrime(b)) R += D[18][a][b];
        }
    }
    cout << fixed << setprecision(10) << R;
}
```

BOJ 1413 박스 안의 열쇠

- 모든 열쇠를 얻기 위해서는 열쇠들끼리 사이클을 구성해야 한다.
  - M개의 폭탄을 이용해서 열쇠를 얻어야 하므로
  - N개의 원소를 M개 이하의 집합으로 나눠서 사이클을 만들면 된다.

  - D(N, M) = N개의 원소를 M개의 원순열로 분할하는 경우의 수
    - 이거 사실 제1종 스털링 수임
  - { D(N, 1) + D(N, 2) + … + D(N, M) } / { D(N, 1) + D(N, 2) + … + D(N, N) }을 출력하면 된다.

- N-1개의 원소가 K개의 사이클을 구성하고 있을 때, N번째 원소를 추가하는 상황을 생각해보자.
  - N번째 원소가 기존 사이클에 들어가는 경우 : N-1개의 간선 중 하나를 선택해서 그 자리에 들어감
  - 혼자서 새로운 사이클을 구성하는 경우

  - D(N, K) = (N-1) * D(N-1, K) + D(N-1, K-1)

BOJ 1413 박스 안의 열쇠

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

ll N, M, D[22][22];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    D[1][1] = 1;
    for(int i=2; i<=N; i++){
        for(int j=1; j<=i; j++){
            D[i][j] = D[i-1][j-1] + (i-1) * D[i-1][j];
        }
    }

    ll A = 0, B = 0, G;
    for(int i=1; i<=M; i++) A += D[N][i];
    for(int i=1; i<=N; i++) B += D[N][i];
    G = __gcd(A, B);
    cout << A / G << "/" << B / G;
}
```

비트필드를 이용한 DP

- DP를 결국 부분 문제의 “상황”를 잘 표현하는 것
  - 원소가 15개 있고, 각 원소를 선택한 경우/선택하지 않는 경우를 나타내야 한다면
  - 15차원 배열 잡는 것보다 비트필드를 이용해서 집합을 표현하는 것이 좋음

  - 문제의 “상황”을 잘 표현하는 방법 중 하나임

BOJ 2133 타일 채우기

- D(N, bit) = 3 * N 타일을 채우는데, N-1번째 열은 다 채워져 있고, N번째 열의 상태가 bit가 되는 경우의 수
  - bit = 0 : N번째 열이 모두 비워져 있음
  - bit = 1 : N번째 열의 첫 번째 행만 채워져 있음
  - bit = 2 : N번째 열의 두 번째 행만 채워져 있음
  - bit = 3 : N번째 열의 첫 번째, 두 번째 행이 채워져 있음
  - …
  - bit = 7 : N번째 열이 모두 채워져 있음

- D(0, 7) = 1로 초기화
- 열심히 그림을 그리면서 점화식을 구하면 된다.

BOJ 2133 타일 채우기

- D(N, 0) = D(N-1, 7)
  - N번째 열이 비워져 있다면 N-1번째 열은 모두 채워져 있어야 함
- D(N, 1) = D(N-1, 6) / D(N, 2) = D(N-1, 5) / D(N, 4) = D(N-1, 3)
  - N번째 열의 첫 번째 행만 채워져 있다면, 가로로 타일을 배치해야 하므로 N-1번째 열의 첫 번째 칸이 비워져 있어야 함
- D(N, 3) = D(N-1, 4) + D(N-1, 7) / D(N, 6) = D(N-1, 1) + D(N-1, 7)
  - N번째 열의 첫 번째 행과 두 번째 행이 채워져 있다면 가로로 타일 2개를 배치(4)하거나, 세로로 타일(7)을 배치하면 됨
- D(N, 5) = D(N-1, 2)
  - 위의 경우와 비슷하지만 세로로 배치하면 안 됨
- D(N, 7) = D(N-1, 0) + D(N-1, 3) + D(N-1, 6)
  - 가로로 3개 배치하는 경우 : D(N-1, 0)
  - 위에 가로로 배치, 아래 세로로 배치 : D(N-1, 6)
  - 위에 세로로 배치, 아래 가로로 배치 : D(N-1, 3)

BOJ 2098 외판원 순회

- D(v, bit) = 방문한 정점들의 집합이 bit이고 현재 v에 있을 때, 남은 정점들을 모두 방문하는 최소 비용
  - v에서 갈 수 있는 정점 중 아직 방문하지 않은 정점 i에 대해
  - D(v, bit) = min{ D(i, bit ∪ i) + C(v, i) }

  - 방문 확인 : bit & (1 << i)
  - 집합에 원소 추가 : bit | (1 << i)

BOJ 2098 외판원 순회

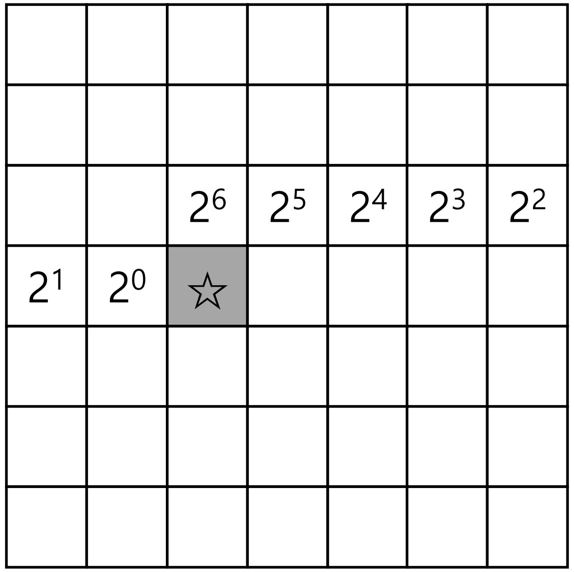
```cpp
#include <bits/stdc++.h>
using namespace std;
constexpr int INF = 0x3f3f3f3f;

int N, C[16][16], D[16][1<<16];

int f(int v, int bit){
    if(bit == (1 << N) - 1){
        if(!C[v][0]) return INF;
        else return C[v][0];
    }
    int &res = D[v][bit];
    if(res != -1) return res;

    res = INF;
    for(int i=0; i<N; i++){
        if(bit & (1 << i)) continue;
        if(C[v][i]) res = min(res, f(i, bit | (1 << i)) + C[v][i]);
    }
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=0; i<N; i++) for(int j=0; j<N; j++) cin >> C[i][j];
    memset(D, -1, sizeof D);
    cout << f(0, 1);
}
```

BOJ 10937 두부 모판 자르기

- (i, j)를 가져가는 경우는 (i-1, j) + (i, j)와 (i, j-1) + (i, j)가 있음
  - (i-1, j) + (i, j)를 가져가기 위해서는 앞에서 (i-1, j)를 가져가면 안 됨
  - (i, j-1) + (i, j)를 가져가기 위해서는 앞에서 (i, j-1)를 가져가면 안 됨

- 이전 N개 칸의 선택 여부를 비트필드로 표현
  - D(i, j, bit) = 현재 (i, j)를 보고 있고, 이전 N개의 칸 선택 여부가 bit일 때, 앞으로 얻을 수 있는 최댓값
  - 만약 2^{N-1}이 꺼져 있다면 D(i, j, bit) ← D(i, j+1, bit << 1 | 1) + C[(i-1, j), (i, j)]
  - 만약 2^0이 꺼져 있다면 D(i, j, bit) ← D(i, j+1, bit << 1 | 3) + C[(i, j+1), (i, j)]
  - (i, j)를 가져가지 않는다면 D(i, j, bit) ← D(i, j+1, bit << 1)

  - bit는 하위 N개 비트만 가져감
  - (0-based) 기저 조건 : i == N
  - (0-based) j == N이면 (i+1, j) 호출



BOJ 10937 두부 모판 자르기

```cpp
#include <bits/stdc++.h>
using namespace std;

constexpr int C[4][4] = {
    { 100, 70, 40, 0 },
    {  70, 50, 30, 0 },
    {  40, 30, 20, 0 },
    {   0,  0,  0, 0 }
};

int N, A[11][11], D[11][11][1<<11], FULL;

int f(int i, int j, int bit){
    bit &= FULL;
    if(i == N) return 0;
    if(j == N) return f(i+1, 0, bit);
    int &res = D[i][j][bit];
    if(res != -1) return res;

    res = f(i, j+1, bit << 1);
    if(i > 0 && !(bit & 1 << N-1)) res = max(res, f(i, j+1, bit << 1 | 1) + C[A[i-1][j]][A[i][j]]);
    if(j > 0 && !(bit & 1)) res = max(res, f(i, j+1, bit << 1 | 3) + C[A[i][j-1]][A[i][j]]);
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++){
            char c; cin >> c;
            A[i][j] = c == 'F' ? 3 : c - 'A';
        }
    }
    memset(D, -1, sizeof D);
    FULL = (1 << N) - 1;
    cout << f(0, 0, 0);
}
```

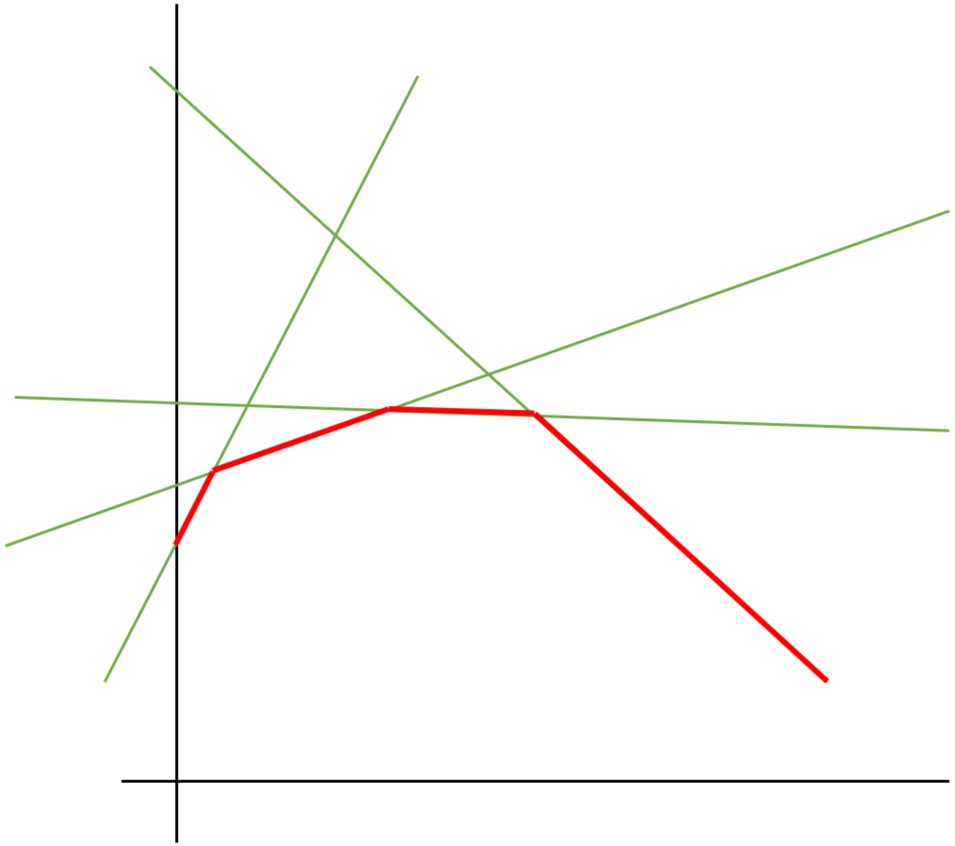
Convex Hull Trick

- 지금까지는 점화식만 잘 찾아서 계산하면 됐지만, 가끔은 점화식을 계산하는 과정도 최적화를 해야 함
- 다양한 테크닉 중 가장 유명한 Convex Hull Trick만 맛보기로 소개

- 점화식이 D[i] = min{ D[j] + A[i] * B[j] } 꼴인 경우
  - min 함수 안에 있는 식은 기울기가 B[j]이고 y절편이 D[j]인 일차 함수
  - 일차 함수가 여러 개 있을 때, x좌표가 주어지면 함숫값의 최솟값을 구하는 문제

Convex Hull Trick

- 최솟값의 형태가 Convex Hull 모양이라서 Convex Hull Trick이라고 부름
- 함수의 기울기가 감소하는 순서대로 주어지고, x좌표는 증가하는 순서대로 주어지면 O(N)에 해결할 수 있음
  - 구체적인 방법은 문제를 풀면서 살펴보자.



BOJ 6171 땅따먹기

- W[i] > W[j] & H[i] > H[j]이면 j는 신경쓰지 않아도 됨
- 직사각형의 너비가 증가하도록(높이가 감소하도록) 정렬하자.
  - 연속한 직사각형은 한 번에 구매하는 것이 이득이다.

- D[i] = 1..i번째 직사각형을 구매하는 최소 비용
  - D[i] = min{ D[j-1] + H[i] * W[j] }
  - Naïve하게 계산하면 O(N^2)이라서 시간 초과

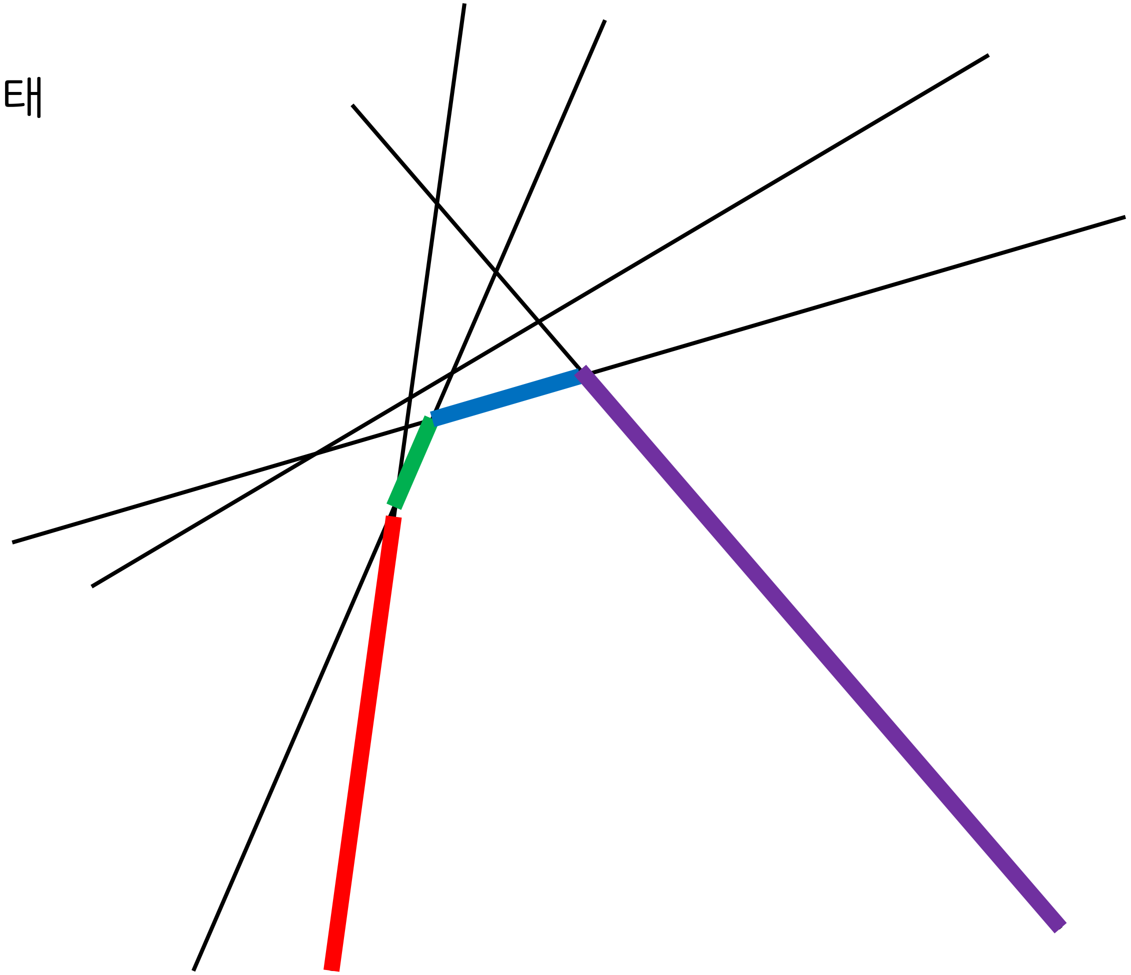
  - W[j] = a, D[j-1] = b, H[i] = x로 치환하면
  - D[i] = min{ ax + b } 이므로 Convex Hull Trick 적용 가능
  - W(기울기)는 감소하고 H(x좌표)는 증가한다.

BOJ 6171 땅따먹기

- 최솟값을 갖는 함수들은 기울기가 감소하는 볼록 함수 형태

- 해야 할 일
  - 최솟값을 갖는 직선들의 집합을 관리
  - 직선 집합에서 최솟값 찾기

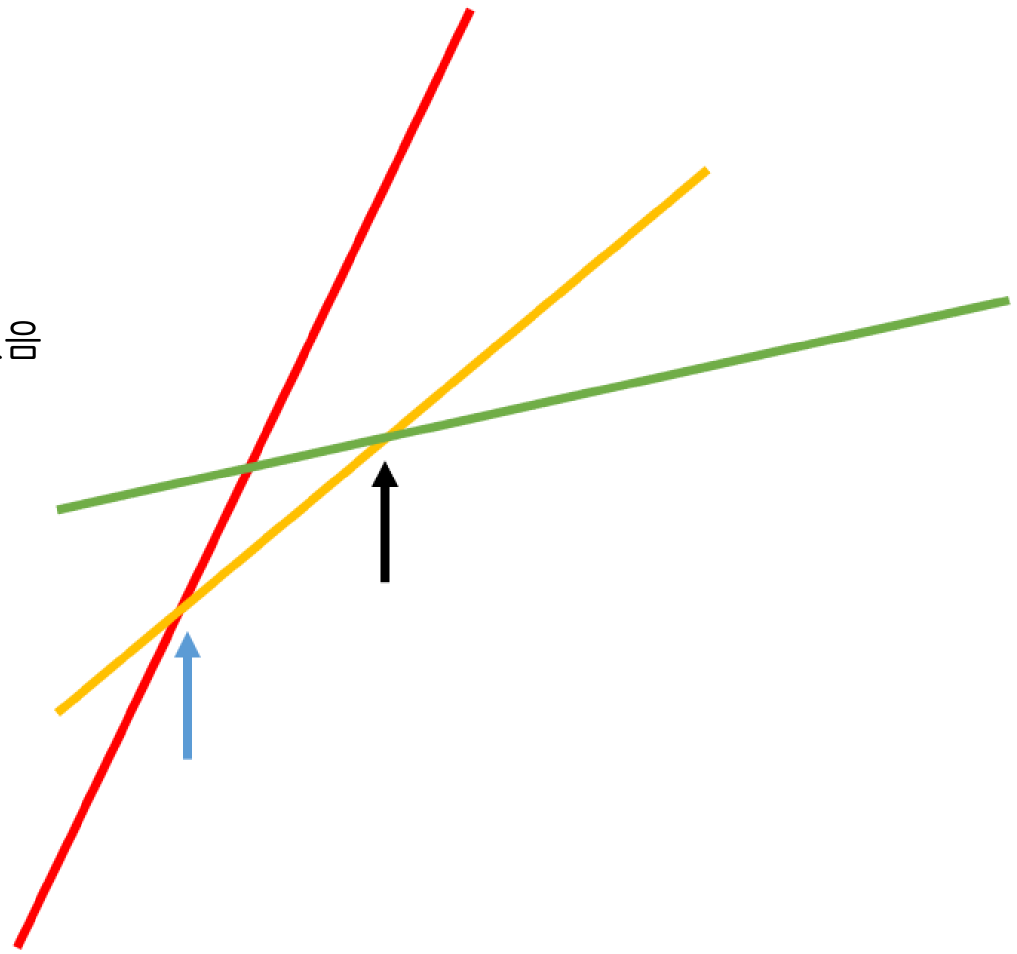
  - 당연히 기울기가 감소하도록 관리



BOJ 6171 땅따먹기

- 최솟값을 찾는 선분 집합 관리
  - 빨간 직선과 노란 직선이 있을 때
  - 초록 직선을 삽입하는 상황

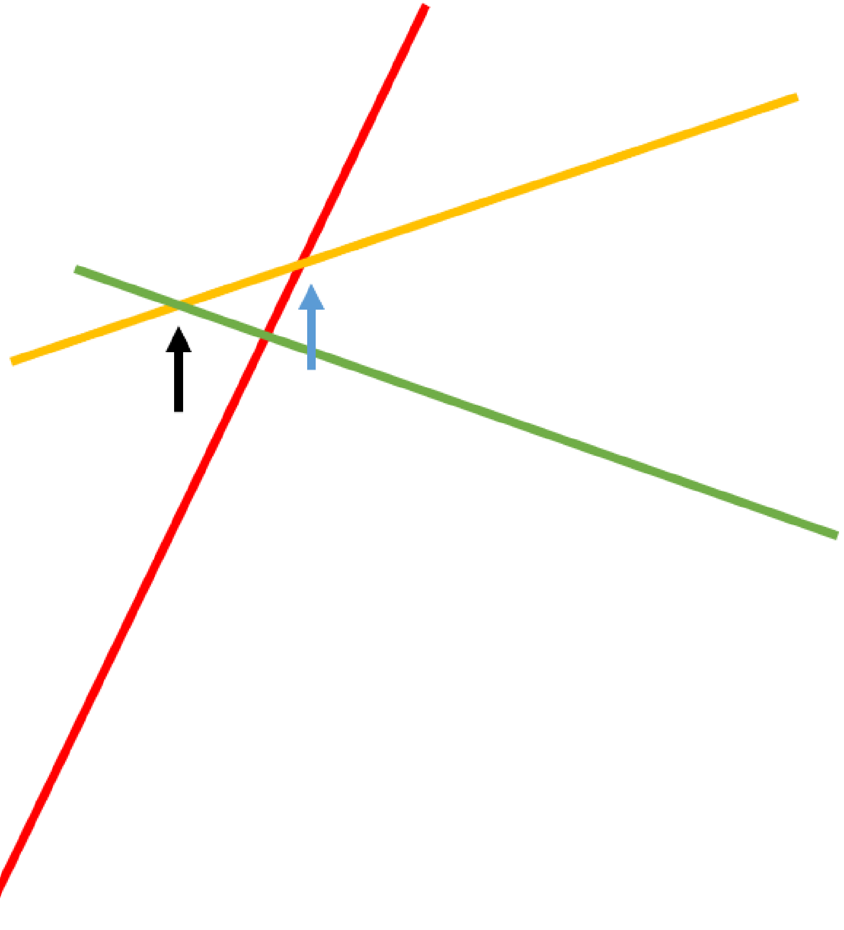
  - 초록 직선이 들어와도 노란 직선이 최소가 되는 구간이 남아있음



BOJ 6171 땅따먹기

- 최솟값을 찾는 선분 집합 관리
  - 빨간 직선과 노란 직선이 있을 때
  - 초록 직선을 삽입하는 상황

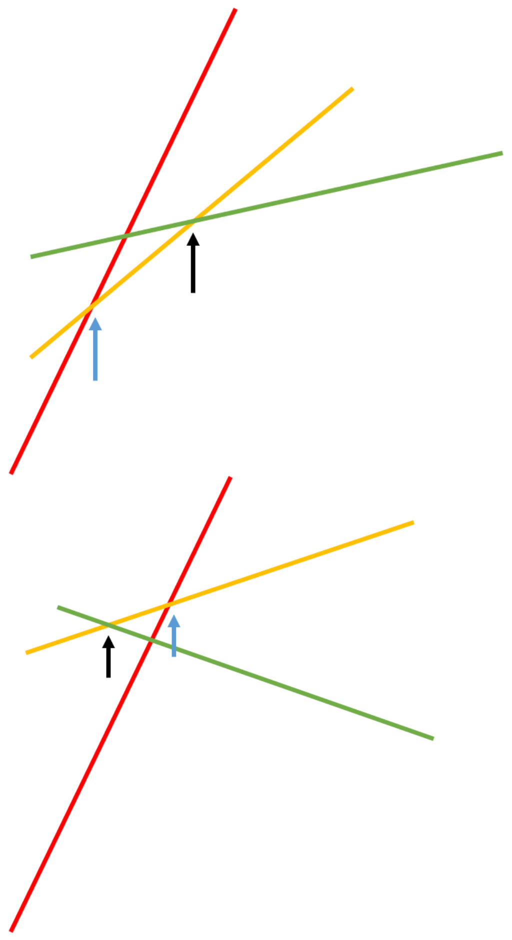
  - 초록 직선이 들어오면 노란 직선이 최소가 되는 구간이 없어짐



BOJ 6171 땅따먹기

- 빨간색, 노란색, 초록색 직선 : 1, 2, 3번 함수
- 1번 함수와 2번 함수의 교점 : x1
- 2번 함수와 3번 함수의 교점 : x2

- x1 < x2이면 2번 함수가 최소가 되는 구간 존재
- x1 ≥ x2이면 2번 함수가 최소가 되는 구간 없음



BOJ 6171 땅따먹기

- 기울기가 감소하는 순서대로 직선이 주어지므로, 직선들을 Graham's Scan처럼 스택으로 관리할 수 있음
  - while(x1 > x2) stk.pop();
  - stk.push(line)
- 기울기의 단조성이 없다면 조금 더 복잡함 (궁금하면 Li-Chao Tree를 찾아보자.)

```
void insert(int a, int b){ // ax+b
  Line line(a, b);
  while(stk.size() >= 2 &&
        cross(stk[-2], stk[-1]) > cross(stk[-1], line)){
    stk.pop_back();
  }
  stk.push_back(line);
}
```

BOJ 6171 땅따먹기

- 주어지는 x좌표에서 최솟값을 찾아야 하고, x좌표는 증가하는 순서대로 주어짐
- 최소가 되는 직선의 기울기가 감소하므로, 스택에서 인덱스를 관리하면 됨
  - i번째 직선과 i+1번째 직선의 교점이 x보다 작거나 같으면 i를 증가시킴
  - 최종적으로 가리키게 되는 직선이 최솟값을 갖게 됨
- 주어지는 x좌표가 증가하지 않는다면 기울기에 대한 이분 탐색을 해야 함

```
int pt;
int query(int x){
  while(pt+1 < stk.size() &&
        cross(stk[pt], stk[pt+1]) <= x){
    pt++;
  }
  return stk[pt].a * x + stk[pt].b;
}
```

BOJ 6171 땅따먹기

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

struct Line{
    ll a, b;
    ll f(ll x){ return a * x + b; }
};
double Cross(const Line &l1, const Line &l2){
    return 1.0 * (l2.b - l1.b) / (l1.a - l2.a);
}

vector<Line> CHT;
int Idx;
void Insert(ll a, ll b){
    Line now = {a, b};
    while(CHT.size() >= 2 && Cross(CHT[CHT.size()-2], CHT.back()) > Cross(CHT.back(), now))
CHT.pop_back();
    CHT.push_back(now);
}
Line Query(const ll x){
    while(Idx + 1 < CHT.size() && CHT[Idx].f(x) >= CHT[Idx+1].f(x)) Idx++;
    return CHT[Idx];
}

int N, M;
pair<ll,ll> A[50505];
ll D[50505];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i].first >> A[i].second;
    sort(A+1, A+N+1);
    for(int i=1; i<=N; i++){
        while(M > 0 && A[M].second <= A[i].second) M--;
        A[++M] = A[i];
    }
    N = M;

    Insert(A[1].second, 0);
    for(int i=1; i<=N; i++){
        auto now = Query(A[i].first);
        D[i] = now.a * A[i].first + now.b;
        Insert(A[i+1].second, D[i]);
    }
    cout << D[N];
}
```

BOJ 20197 3D Histogram

- https://github.com/justiceHui/Sunrin-SHARC/blob/master/2020-2nd/SHARC_Hard_Problem.pdf
- 풀이는 어렵지 않은데 구현이 귀찮음

더 공부할 거리

- **문제 많이 풀기!**
- 선형 점화식의 빠른 계산
  - 행렬 이용 : O(K^3 log N)
  - Kitamasa Method : O(K^2 log N)
  - Kitamasa + FFT : O(K log K log N)
- 점화식이 특정 조건을 만족할 때 빠르게 계산하는 방법
  - Divide and Conquer Optimization
  - Monotone Queue Optimization
  - Aliens Trick
  - Slope Trick
  - Knuth Optimization
- 점화식의 상태를 잘 나타내는 방법
  - Connection Profile DP
  - Sum Over Subsets DP (SOS DP)